5-2017

Storing and Rendering Geospatial Data in Mobile Applications

Samip Neupane
University of New Orleans

Storing and Rendering Geospatial Data

in

Mobile Applications

An Honors Thesis Submitted to the Faculty of

the Department of Computer Science of

the University of New Orleans

In Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science, with University High Honors

and Honors in Computer Science

By

Samip Neupane

May 2017

Acknowledgement

I would like to thank Dr. Mahdi Abdelguerfi for providing me with the opportunity to work on this project under his guidance as well as providing me with the proper resources and advice for this project. I would also like to thank Nathan Cooper and Dr. Elias Ioup for their input and guidance in the project. Finally, I would like to Brent Barre and Dr. Christopher Summa for their help in the process of developing the application and completion of this thesis.

Table of Contents

Table of Figures

Abstract

Geographical Information Systems and geospatial data are seeing widespread use in various internet and mobile mapping applications. One of the areas where such technologies can be particularly valuable is aeronautical navigation. Pilots use paper charts for navigation, which, in contrast to modern mapping software, have some limitations. This project aims to develop an iOS application for phones and tablets that uses a GeoPackage database containing aeronautical geospatial data, which is rendered on a map to create an offline, feature-based mapping software to be used for navigation. Map features are selected from the database using R-Tree spatial indices. The attributes from each feature within the requested bounds are evaluated to determine the styling for that feature. Each feature, after applying the aforementioned styling, is drawn to an interactive map that supports basic zooming and panning functionalities. The application is written in Swift 3.0 and all features are drawn using iOS Core Graphics.

Key Words: GeoPackage, SQLite, iOS, Aeronautics, Feature Mapping, R-Tree

Introduction

Traditional navigation systems used by US military pilots relied on paper charts and platform based software. Hardware limitations within the aircraft and the physical size and weight of printed charts limit the amount of airspace coverage that pilots can bring onboard a flight. Flight Information Publications (FLIPs) are paper navigation charts that pilots have been using for decades. They depict air traffic service routes, waypoints and airport, for both high and low altitude. It takes 48 total FLIPS to cover the entire US. It is quite difficult to carry all these on board in the pilot's flight bag. Additionally, the charts are created at two map scales and the information in busy areas is often cluttered and difficult to read (Barré et al. 2016). To overcome this, a single worldwide, digital chart was created: the Seamless Enroute chart. The Naval Research Laboratory Geospatial Computing section first developed the Seamless Enroute chart for use in web-based applications. It was designed to closely reproduce the appearance of FLIPS but as a worldwide interactive chart. The goal of the project described in this thesis is to produce a mobile implementation of the Seamless Project. As pilots are now carrying tablets and handheld devices for use in the cockpit, the mobile adaptation of Seamless Enroute has the potential to revolutionize in-flight mapping for military pilots.

Mobile Seamless can be an effective substitute for FLIPS. A map of a specific route can be loaded in the device prior to takeoff. This provides a full interactive use of the chart on mobile devices even without an internet connection. Mobile Seamless is a feature-based iOS mapping application based on a GeoPackage database.

GeoPackage is an open, standards-based, platform-independent, portable, self-describing, compact format for storing and transferring geospatial information implemented as a SQLite database container. The database contains feature tables, whose data represents entities having different geometry types including, but not limited to, lines, polygons and points. Features are homogeneous collections of common entities, each having the same spatial representation and a common set of attribute columns. Each feature is stored in an individual table and all feature tables contain a geometry column, which stores the geometry type and position of the feature. (GeoPackage Encoding Standard)

Features are selected from the GeoPackage database using an R-Tree as a spatial index. R-trees are tree data structures for indexing multidimensional information. In our mobile application, features are queried from the R-Tree tables, which return all the features that are completely inside the given bounding box. The features are then displayed on a map view generated using Swift's UIView. The map supports zoom and pan features through buttons. If users zoom or pan, the bounding box changes, which results in the R-tree query using the new bounds, which returns new set of features inside the current bounding box.

After obtaining the features, we analyze each feature attributes and apply styles to the features based on data stored in the XML file. The XML files are divided into two categories: *stylesheet* and *stylemap*. The stylemap contains information necessary to select appropriate stylesheet for features based of their attribute values. The stylesheet contains the customized designs to be

applied to each feature. Customization information is specific to symbol type: lines have their own style while points and areas have different custom characteristics. After getting the appropriate style information, features are drawn on a map by applying the styles. As mentioned earlier, Mobile Seamless uses feature-based rendering.

Feature-Based and Tile-Based Rendering

The process of rendering a map generally means making a visual map from raw geospatial data. Two of the most common techniques for map rendering are feature-based and tile-based rendering. This project uses feature-based map rendering; however, both map rendering techniques are discussed below.

Feature-Based Rendering

Features are collections of geographic entities with a certain geometry type (e.g. point, line, or polygon), attributes, and a spatial reference. As such, a feature is an abstract representation of a real-world object on a map. Roads and rivers, for example, can be represented by a line feature; likewise, airports can be represented by point features. A feature table contains data related to features along with information that helps us describe the feature. The feature's coordinates are used to place the feature on a map, and the feature's attributes are used to define the visual styling/symbology of the feature, which is drawn upon request.

Tile-Based Rendering

Tile-based maps consist of many individually-requested tile image files. Each zoom level contains a set of images that are arranged together to display the level of detail specific to that level. Tile based rendering is sometimes faster than feature based rendering because, unlike

feature-based mapping, there is no heavy computation involved as there is no need to draw individual feature; the image tiles are pre-rendered and simply fetched when requested. It is currently the most popular way to display and navigate maps.

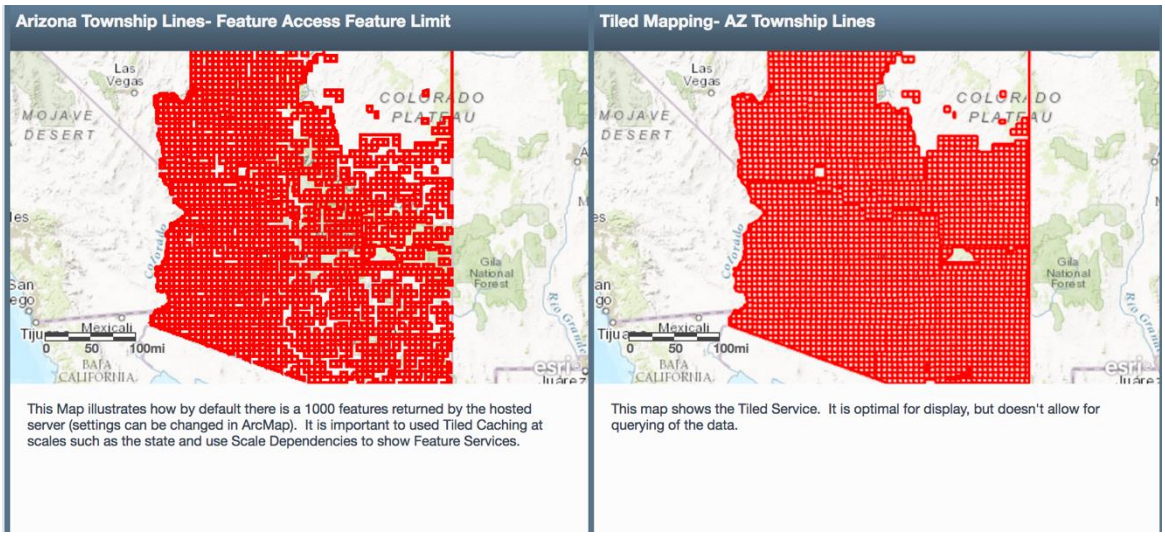

Figure 1: Feature Based Mapping vs Tile Based Mapping (Arizona Township Lines-Feature Access Feature Limit, Web)

The above figure shows the difference between feature-based mapping and tile-based mapping. Feature mapping uses drawing functionality provided by the system to draw and style shapes while tile mapping splits the map up into a pyramid of images at multiple zoom levels. Both techniques have their own advantages and disadvantages and it is useful to consider both of these techniques while working with map rendering.

Components of the Mobile Seamless Application

GeoPackage

Mobile device users who require map/geospatial application services and operate in disconnected or limited network connectivity environments are challenged by limited storage capacity and the lack of open-format geospatial data to support these applications. The GeoPackage standard defines the schema for table definitions, integrity assertions, format limitations, and content constraints. It provides a single file suitable for disconnected mobile devices, and is "serverless" with clients accessing the file directly through an API (OGC® GeoPackage Encoding Standard).

Despite the simple single-file model, GeoPackage considers itself a Relational Database Management System and supports transactional features that guarantee that all changes to data in the container are Atomic, Consistent, Isolated, and Durable (ACID) despite program crashes, operating system crashes, and power failures (GeoPackage Encoding Standard). GeoPackage can be used to store both tile-based and feature-based data, and so it supports both rendering techniques discussed in the previous section.

The database used in this project ("aero.gpkg") contains feature data that correspond to real world geographical entities. There are several geometry tables in the database, each of which contains features that can be one of either points, polygons, or lines. Points, for example, may correspond to airports or radio towers. Lines correspond to airplanes routes; polygons correspond to a variety of geographical entities like airspaces and air traffic control areas. Each

geometry table has a *gpkg_id* which is the primary key for the table. Each row from the geometry table is stored as an Attribute class in the application, which contains a hash map containing the column name and the value. These values are used to determine the custom styles to be applied to the feature represented by the row.

The database also contains R-Tree tables, which correspond to each of the geometry tables. R-Tree is a tree data structure, similar to a B-tree, used for indexing spatial data within a database. In an R-Tree structure, data is sorted into a set of hierarchical nodes that may overlap. Each node has a variable number of entries, each of which includes an identifier for child nodes or actual data elements and a bounding box for all entries within the child node or the data elements. Searching algorithms check the bounding boxes before searching within a child node, thus avoiding extensive searches (Esri Support GIS Dictionary). In order to get features from the geometry table, we query the R-Tree that corresponds to the table. This returns all the nodes that are in the bounding box along with their *gpkg_id*. The *gpkg_id* can be used to get all the columns from that specific geometry (feature) table. Feature and Attribute classes are created in the Mobile Seamless app based on these values. The figure below displays the function that returns all features from the bounds for a specific table. The function takes a table name and the bounding box information (minX, maxX, minY, maxY) as parameters and returns an array of hashmap. The hashmap contains all the key-value pairs for a single row.

```
public func getFeaturesFromBounds(tableName: String, minX: Double, maxX: Double, minY: Double,
    maxY: Double)-> [[String:Any]]{
    var allFeatures: [[String:Any]] = []
    let columns = "*"
    let featureDatabaseObject = geoPackage.getFeatureDao(withTableName: tableName)
    let query: String = "Select \(columns) from \(tableName) where gpkg_id in (SELECT id FROM
        rtree_\(tableName)_geometry WHERE minX>=\(minX) AND maxX<=\(maxX) AND minY>=\(minY)  AND
        maxY<=\(maxY))"
    let resultSet: GPKGResultSet = (featureDatabaseObject?.rawQuery(query))!
    while(resultSet.moveToNext()){
        var hash: [String: Any] = [:]
        let featureRow: GPKGFeatureRow = featureDatabaseObject!.getFeatureRow(resultSet)
        let columnNames: [String] = (featureRow.getColumnNames() as! [String])
        for column in columnNames{

            let value  = (featureRow.getValueWithColumnName(column))
            hash[column] = value
        }
        allFeatures.append(hash)

    }
    resultSet.close()
    return allFeatures

}
```

Figure 2: Function that returns all the features in the given bound for a database table.

The geopackage database used in the project was parsed using a third-party CocoaPod named *geopackage-ios*. A CocoaPod is a general term for either a library or framework that's added to the project by using the CocoaPods tool. *GeoPackage-ios* provides an interface to access the geopackage database and allows us to use the R-Tree indexes. Initially, we started by writing our own geopackage parser, but later, a pre-existing robust iOS library was found, which we used instead. The pod can be found at ngageoint/geopackage-ios on github (Bosborn).

XML

Another important piece of the puzzle is the XML. Extensible Markup Language describes a class of data objects and partially describes the behavior of computer programs which process them.

XML is an intermediate, platform independent format, so it is widely used for data transfers. There are three major XML components in this project, namely *StyleSheet, StyleMap* and *BaseMapConfig*. XML files are used as configuration files for the map rendering code and as containers for storing style information to be applied to selected features.

The StyleMap element contains a LayerMapping element. The LayerMapping element contains a layer name corresponding to the GeoPackage table, a DefaultStyle element and a ConditionalStyle element. ConditionalStyle contains arrays of AttributeExpression elements, which contain a key-value pair that represents an attribute name and its value. The value of attributes from features obtained from the GeoPackage is compared to the value of attributes inside AttributeExpression and, if they match, a specific stylesheet name to be applied to the features is selected. If none of the attribute values satisfy the conditions, a default style is selected. The DefaultStyle element contains the default style name to be applied if the attributes from the features do not satisfy the conditions inside any of the ConditionalStyle elements. In order to select the style for a feature, the attributes for a feature are queried. The query returns a hashmap mapping the attribute names to their values. An AttributeExpression element in StyleMap contains an attribute name and value pair. If the key-value pairs in a particular AttributeExpression match the key value pairs in a given GeoPackage feature's attributes, then the corresponding ConditionalStyle is used to render the feature. The following figure is a code snippet from the StyleSheet.

```
<LayerMapping layerName="ATS_ROUTE_LOW">
    <DefaultStyle name="default"/>
    <ConditionalStyle name="ats_advisory">
        <AttributeExpression attribute="ATS_TYPE" operator="equal" value="D"/>
    </ConditionalStyle>
    <ConditionalStyle name="ats_advisory">
        <AttributeExpression attribute="ATS_TYPE" operator="equal" value="C"/>
    </ConditionalStyle>
    <ConditionalStyle name="ats_rnav">
        <AttributeExpression attribute="ATS_TYPE" operator="equal" value="R"/>
    </ConditionalStyle>
    <ConditionalStyle name="ats_lf_mf">
        <AttributeExpression attribute="FREQ_CLASS" operator="equal" value="B"/>
    </ConditionalStyle>
    <ConditionalStyle name="ats_vhf_uhf">
        <AttributeExpression attribute="FREQ_CLASS" operator="equal" value="A"/>
    </ConditionalStyle>
</LayerMapping>
```

Figure 3: Content of XML StyleMap that shows how elements are organized in it

The above figure shows different elements from a StyleMap and how these elements are organized. As shown here, the element LayerMapping contains DefaultStyle and ConditionalStyle elements. The ConditionalStyle element further contains AttributeExpression, which contains an attribute name, operator and value. While parsing the XML, all this information is stored in a Swift class. The *layerName* corresponds to the table name that is currently being used. A Feature class derived from the database contains attributes which are compared against the attributes in the ConditionalStyles. For example, if the current table being queried is ATS_ROUTE_LOW and its attribute has a key “ATS_TYPE” and value “C”, the style named “ats_rnav” is selected based on the attribute key and value in AttributeExpression. If none of the feature attributes match, a default style name is returned.

The StyleSheet contains visual information to be applied to the features. It contains several FeatureStyle elements. The style name selected as described in the previous paragraph is used to select one of these FeatureStyles. Each FeatureStyle can have text and symbol elements.

Information describing how to visually depict lines, areas, and points is stored inside symbol elements. Each symbol has a priority which defines the precedence of the symbols. It defines the z-order while drawing symbols and determines which feature should be on top when two symbols overlap. Each symbol also has a Symbol Scale Range that contains the minimum and maximum scale for which the symbol should be visible. The Text element inside the FeatureStyle element contains text definitions (font, size, color, etc.) for styling the label that will be placed near the symbol. The figure below is a code snippet from a stylesheet XML.

```
<FeatureStyle name="ats_rnav">
    <Symbols>
        <SymbolScaleRange minScale="4000000" maxScale="1000000000000">
            <Lines>
                <Line priority="7">
                    <LineColor>#0850A3</LineColor>
                    <LineWidth>1</LineWidth>
                    <LineCap />
                    <LineJoin />
                    <LineOpacity />
                </Line>
            </Lines>
        </SymbolScaleRange>
        <SymbolScaleRange minScale="2000000" maxScale="4000000">
            <CustomLine priority="6">
                <PackageName>vectorvisualization.aero</PackageName>
                <ClassName>AeroDrawableLineFeature</ClassName>
            </CustomLine>
        </SymbolScaleRange>
        <SymbolScaleRange minScale="0" maxScale="2000000">

            <CustomLine priority="6">
                <PackageName>vectorvisualization.aero</PackageName>
                <ClassName>AeroDrawableLineFeature</ClassName>
            </CustomLine>
        </SymbolScaleRange>
    </Symbols>
    <Text>
        <TextScaleRange minScale="0" maxScale="1000000000000">
            <LineText>
                <TextDefinition size="12" font="SANS_SERIF" color="#0850A3" style="BOLD">
                </TextDefinition>
            </LineText>
        </TextScaleRange>
    </Text>
</FeatureStyle>
```

Figure 4: Content of a FeatureStyle showing how elements are organized in it

The figure above shows a FeatureStyle element. The name of this FeatureStyle is “ats_rnav”, which was selected by the Stylemap in figure 3. The Symbols element shown above contains point, line and area elements. Each of these elements contains different styling information inside it. If this particular FeatureStyle is selected for a line feature, then the line’s color would be #0850A3 and its width 1.0. If the FeatureStyle is selected for a text feature, then the text size would be 12, the font would be “SANS_SERIF”, text color would be #0850A3 and the text style would be bold. Similarly, if this FeatureStyle is selected for a CustomLine, the class inside ClassName element would contain the style information. This is how information present in an XML is used to style the features. As discussed above, symbols contain a minScale and maxScale which gives information about the zoom levels at which these features should be displayed. In addition to the above elements, symbols might contain other elements including shapes, custom lines and custom area all of which are used to provide custom styles to the features.

The last XML file to discuss in this project is the BaseMapConfig, which contains information that links map zoom levels to data table/layer names. This allows us to select tables based on the current, user-controlled map scale level. For example, if a table contains data that the developer has deemed less important, it should be displayed only after a certain zoom level when it will not interfere as much with more-important data. BaseMapConfig.xml contains information that allows us to use the scale values generated using the map’s current bounding box to select the data tables that are visually desirable for that scale. This helps features to be drawn only when they are properly spaced apart and prevents us from drawing all the features into a cluttered mess.

The XML components described above were already in place for the original Seamless chart. However, the project for this thesis, being a mobile version of Seamless, required several adaptations. Several Swift classes based on the XML schema (.xsd) were written to parse the XML files. The function “parse” in class StyleMapParser takes a XML file as argument, parses and returns an instance of the StyleMapper class that contains all the information stored in the XML document. Similarly, a function “parse” in StyleSheetParser returns an instance of MMSSMapMakeStyleSheet, which also contains all the information in the XML documents by creating appropriate classes to hold relevant information. Instances of StyleMapParser and MMSSMapMakerStyleSheet are passed to a function “draw” in MapMaker where feature attributes are evaluated and appropriate styles are applied to features. Parsing the XML was done using a pod named SWXMLHash (Drmohundro). SWXMLHash was used because of its relatively simple interface that made parsing XML file easier and efficient.

Map Interface

In conventional GIS terms, a basemap is the background setting for a map. It depicts background reference information such as landforms, roads, landmarks, and political boundaries, onto which other thematic information is placed. (Esri Support GIS Dictionary) BaseMaps also add aesthetic appeal to maps.

On the other hand, NRL uses the term *basemap* in a slightly unconventional way when it comes to their own map products. Here, a basemap is a single map layer that dynamically

includes/excludes data from multiple data layers based on scale and clutter, as described in the previous section. The mobile application for this thesis adopts this usage of a basemap, using it as the one, standalone, dynamic layer displayed. Therefore, it is not so much a background as it is the focus – the basemap in this case is the complete map itself. The contents and scale-dependent nature of the basemap are defined in XML configuration, namely, in BasemapConfig.xml. This file contains the configuration information for the interactive map viewer. It gives information about features that should be displayed within a scale range. This ensures that only specific tables are queried at a given instant, rather that probing all the tables and iterating through them.

Interactive Map Viewer

The Interactive Map Viewer provides a platform for feature rendering. It serves as a background setting for the map and provides necessary detail to orient a location in the map.
The map viewer was created from the ground up using Swift's UIView and UIImageView. The UIView class defines a rectangular area on the screen and the interfaces for managing the content in that area. A UIImageView is an object that displays a single image or a sequence of animated images in your interface. Drawing is done using a Core Graphics Context (CGContext). Features displayed in an image are created using Swift's Core Graphics Context and applied to the UIImageView, which is then added to the UIView.

This approach also needed its own implementation of a standard geographic coordinate system. The leftmost point at the top represents a latitude of +90 and longitude of -180.

Similarly, the rightmost point at the bottom represents a latitude of -90 and a longitude of +180. So, the four corners in the view represent the world coordinates when zoomed out. The bounds change when zoomed or panned. The width and height of the bounding box decrease when zoomed in and increase when zoomed out. Panning shifts the bounds towards either the X or Y axis depending on the direction the image is panned. After each pan or zoom, the image from the previous level is cleared by setting the image inside UIImageView to nil. Afterwards, features are redrawn on CGContext and displayed in the UIView used the same process described above. Currently, buttons are used for panning and zooming.

The Swift class MapConfig contains necessary information related to the map viewer such as information about the bounding box and codes for zooming and panning. It acts as a delegate that organizes the map environment after each zoom or pan event. One of the functions it contains is “getPixelFromXY”. This function returns the X-Y pixel value from longitude and latitude calculated based on the current bounds and zoom level. This is how the geographical data is positioned on the map display. The class also contains an inverse function, named “getXYfromPixel” that returns the latitude and longitude values based on current pixel values and the bounds. This function is used to display the latitude and longitude coordinates when the user taps on the view.

```
public func getPixelFromXY(x: Double, y: Double)-> CGPoint{
    let viewWidth: Int = 364
    let viewHeight: Int = 260

    let xScale = Double(viewWidth) / self.width
    let yScale = Double(viewHeight) / self.height

    let deltaY = maxY - y
    let deltaX = x - minX

    let xN = deltaX * xScale
    let yN = deltaY * yScale

    return CGPoint(x: xN, y: yN)
```

Figure 5: Function that returns the current pixel value for any latitude and longitude

```
public func getXYfromPixel(x: Double, y: Double)-> [Double]{
    let viewWidth: Int = 364
    let viewHeight: Int = 260
    let xScale = Double(viewWidth) / self.width
    let yScale = Double(viewHeight) / self.height

    let deltaX: Double = x / xScale
    let deltaY: Double = y / yScale
    let yRet: Double = maxY - deltaY
    let xRet: Double = minX + deltaX

    return [xRet, yRet]
```

Figure 6: Function that reverses getXYfromPixel. It returns the latitude and longitude value for any pixel value within the current bounds

Connecting the Dots

The mobile app consists of three major inputs: The GeoPackage (the data), the XML configurations that match data features to visual styles, and the XML basemap configuration which defines which data layers are shown at which scales. They are linked in the MapConfig and MapMaker classes.

Initially, the database is queried using a specific bound as parameter for R-Tree indexes. The output from the query is stored as an array of feature iterators, one feature iterator for each table/layer. The feature iterators are passed to a function “draw” in MapMapker.swift. Here, each of these features is evaluated, which involves finding the right style (graphical information), getting the correct pixel values, and calling the appropriate feature renderers. Renderers for points, lines, and areas then render features on the view. Renderers contain specific codes to draw the feature. For example, if the feature is a line, the function “draw” calls LineFeatureRenderer, which contains functions that draw lines on CGContext. Areas and points and handled similarly. Features are drawn to the view based on their scale ranges. The current map scale is calculated using the ratio of the map’s current geographical bounds size to the map display’s fixed pixel size. Any symbol whose scale range does not include the current map scale does not get drawn.

Mobile Seamless, being a fairly large project, needed to be discretized into different phases. Adhering to a timeline, we solved different parts of the project and kept building on to the

progress. Finally, all the individual components were integrated to develop a functional application.

Stages of Evolution: Timeline

The Initial Stage

Mobile Seamless is based on the project Seamless, which is a similar map-rendering application written in Java. The original Seamless uses a different database to store features and runs on map clients like Gaia or QGIS. Mobile Seamless was written in Swift3.0 from ground up using the Java implementation as reference. The initial phase of the project was exploring the GeoPackage Specification. A detailed analysis of the GeoPackage specification provided insights on how data and metadata were stored in a GeoPackage, including ways to parse and query them. The majority of time in the initial phase was spent writing code to parse the GeoPackage, which was rebuffed when we found an iOS library for parsing GeoPackage.

The library provided an easy interface to use swift's MapKit (a Swift framework for displaying map or satellite imagery), which enabled us to render the features on different types of map views provided by MapKit. One of our concerns was the ability of the parser to allow us to use R-Tree spatial indexes. Running test methods provided in the geopackage spec helped us verify the correct implementation of spatial indices on feature table geometry columns. At this point, we had an application that could take a geopackage file and display all the features on a map view provided by MapKit.

Divide and Conquer

After getting the required information from the database, we started to explore different ways to parse the XML. We wanted to integrate XMLs from the original Seamless project so that appropriate styles could be applied to the features. Considering the relatively substantial amount of complexity required for the application to parse the GeoPackage and render the features on map, a separate dummy project was created just to parse the XMLs. Dividing the projects into simpler chunks helped with simplicity and organization. Dummy data was provided for values from the database in order to create a Feature class, which was used to select a specific FeatureStyle. Parsing was done using an external library. After getting all the services setup in the dummy project, it was then integrated to the main project where actual feature attributes from the GeoPackage were used to select FeatureStyles. At this point, we could get all features from the database, and apply styles to them, but we could not render the styled features on a map.

The Final Puzzle

Until now, features were rendered in a map interface provided by Swift using *geopackage-ios*. It was about time to create our own interactive and customizable map viewer. So, a new dummy project was created once more to set up a map viewer. A couple of ideas were analyzed in order to set up the structure of the map. Initially, UIScrollView was used to set up the map. After some deliberation and difficulty to render features on UIScrollView, we tried using UIImageView and UIView to organize the map. Eventually, we were able to simulate a map interface and support zooming and panning. Converting the latitude and longitude value to

pixel value was tricky. After doing some research on basemaps, we could get the zoom and pan working. A feature that allowed us to get the latitude and longitude on a given point in the UIView by tapping at the location was added. After the map was configured, all three features were integrated into a single project and feature renderer classes were written for each feature type (line, point, area, text) in order to display the styled feature of the map.

Currently, work is being done on further enabling the basemap functionality: using feature scale and priorities to query certain tables and only display features that correspond to the scale at the current user-controlled scale level.

The figure below contains screenshots from the Mobile Seamless application. Currently, only symbols (lines, points and polygons) can be displayed in the map. Text labels have not yet been rendered.

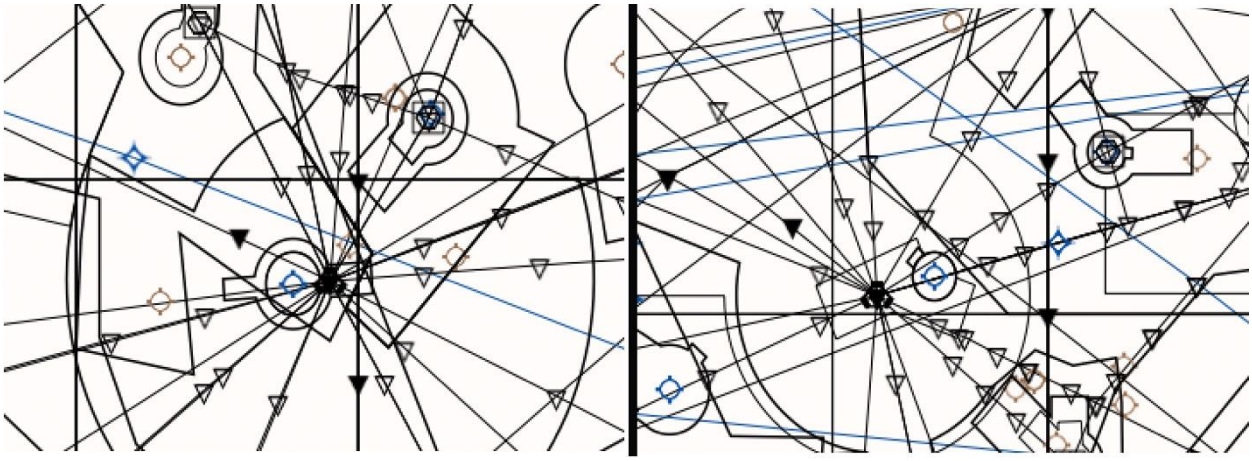

Figure 7: Screenshot from the mobile seamless application

The figure below contains a screenshot from the Seamless Enroute chart.

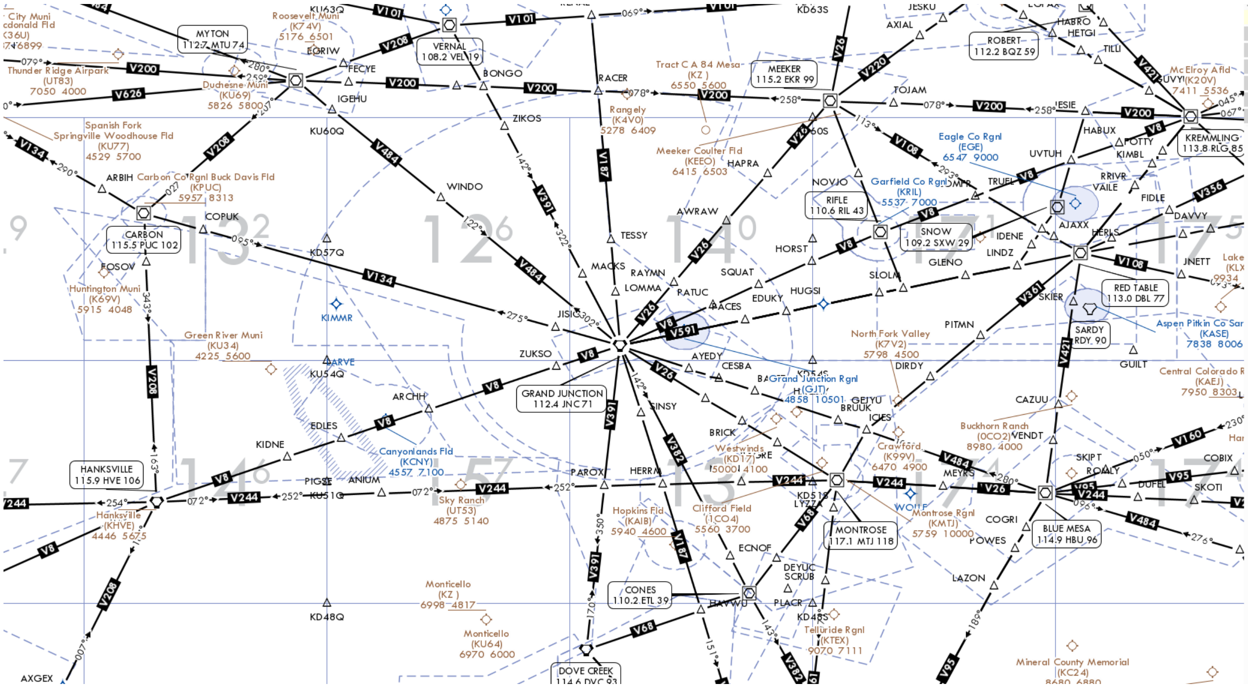

Figure 8: Screen Shot from Seamless run on a map-server

Project Complications and Solutions

Liftoff Setback

Project Seamless required fairly extensive coding and contains a lot of parts, so getting a good grasp on the project was a challenge. Even after studying the code for weeks, it was difficult to understand the logic and flow of execution of the components. After some consulting and exploration, it was thought best to discretize the project into components and tackle one problem at a time. Once broken down, a timeline was created in order to track progress. The initial phase of the project comprised of writing codes for parsing the geopackage, which was

quickly rebuffed when a better alternative was found. The project had a shaky start, but eventually the project headed to the right direction.

The Turbulent Coupling

There are two external libraries (Pods) used in the project; *geopackage-ios*, a pod written in Objective-C and SWXMLHash written in Swift. Parsing the GeoPackage in one dummy project and XMLs in another did not have any complications, but integrating them was an issue. Pods written in Objective-C are generate static libraries while swift pods generate frameworks. There was an issue with *geopackage-ios* while using it with a pod written in swift. It took quite some time to get around this obstacle. The developer of the pod was contacted about the issue, who then released a beta version of pod with the fix. Even after using the beta pod, there was a "Header not found" error with the project. After some more research, a python script (import_header.py) that copied all the objective-c headers in the project to pods/headers folder was written, which finally solved the issue.

Future Work

In order to improve the application and provide users with better tools, we plan to implement certain features in the future. One of them is rendering text based features in the map. Until now, only symbols can be rendered in the map, but we plan to incorporate text rendering soon. Another area where we might need some work done is on implementing more complex custom styles that are defined in Swift classes. As discussed above, most of the feature styles are defined in XML as attribute values, but some of the features need to be defined in custom

classes containing more complex/varying style information and logic that would be difficult to enumerate in XML. Storing the styling information in classes will help customize the features by defining more specific drawing instructions. Although a little restrictive, Swift reflection might be handy while implementing this feature.

Currently, a feature gets rendered on the map view only if it is completely inside a bounding box. For example, if one of the points of a line is inside the bounding box and another point is outside the current bounding box, the feature is not drawn. It only gets drawn when both the points are inside the bounding box, so currently if you pan the map, lines appear on the screen arbitrarily. In order to solve this problem, we initially tried increasing the bounding box width and height while querying for features. However, since the lines are often airplane routes which tend to be long, simply extending the bounding was not be a good solution. A better algorithm needs to be sought for fixing this issue.

Another area that might need improvement is the user interface. Currently, zooming and panning are done through buttons. We plan to make the map interface more intuitive by implementing dragging and pinching abilities. We also plan to use a table view, which would be displayed at the start of the application. The table would contain GeoPackage database names from which, users could select any one the databases to view on the map. This would be handy for pilots to select specific database based on the current flight route. In addition to that, we plan to implement a function that would allow us to remotely download the database and add

it to the table discussed above. The GeoPackage database location is hard-coded in the program right now.

Conclusion

Mobile Seamless was written to display aeronautical charts for pilots on a mobile platform. It allows pilots to use an interactive application instead of physical maps, which can be onerous to handle. Based on a Java project named Seamless, Mobile Seamless is an iOS application for feature based map rendering using GeoPackage as the underlying database. Written in Swift 3.0, it incorporates most of the functionalities of its precursor. The application uses an R-Tree spatial index to select different features from the database. The attributes from the features are then evaluated and matched to the appropriate visual representation. Mechanisms to draw different feature geometry types are set up and features are drawn in the map. As of now, the app is interactive and draws lines, areas and points with custom styles applied to them, but there are still remaining features, which we plan to complete in the near future.

Sources

"Arizona Township Lines- Feature Access Feature Limit." *Arcgis.com*. Arcgis, n.d. Web. 15 Mar. 2017.

Barré, B., N. Schoenhardt, and E. Ioup. "The Digital Seamless Enroute Chart for Aeronautical Data." *2016 NRL Review* n.d.: 136-38. Web.

Barré, Brent. "Personal Interview." Personal interview.

Bosborn. "Ngageoint/geopackage-ios." *GitHub*. N.p., 09 Feb. 2017. Web. 1 Mar. 2017.

Drmohundro. "Drmohundro/SWXMLHash." *GitHub*. N.p., 01 Feb. 2017. Web. 3 Mar. 2017.

"Esri Support GIS Dictionary." *Esri Support GIS Dictionary*. Esri, n.d. Web. 7 Mar. 2017.

"GeoPackage Encoding Standard." *GeoPackage Encoding Standard | OGC*. Open Geospatial Consortium, n.d. Web. 10 Mar. 2017.

"OGC® GeoPackage Encoding Standard." *OGC® GeoPackage Encoding Standard*. Open Geospatial Consortium, n.d. Web. 12 Mar. 2017.

"UIImageView." *UIImageView - UIKit | Apple Developer Documentation*. Apple Inc, n.d. Web. 3 Mar. 2017.

"UIView." *UIView - UIKit | Apple Developer Documentation*. Apple Inc, n.d. Web. 4 Mar. 2017.